UCRL-ID-150000

RPF: An Extensible, Cross-Platform, Binary File Format for Radiation Physics Data

A Compendium of Technical Work papers

Cheryl L. Ham

September 10, 2002

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

University of California
Lawrence Livermore National Laboratory
Technical Information Department
Livermore, CA 94551

RPF: An Extensible, Cross-Platform, Binary File Format for Radiation Physics Data

A Compendium of Technical Work Papers

Cheryl L. Ham

Lawrence Livermore National Laboratory

Contents

Introduction

Lawrence Livermore National Laboratory's Radiation Technology Group (RTG) uses a number of computer codes for simulation and analysis of radiation data. The number of incompatible data formats that these data presented themselves in have continued to multiply. In the 1980's a Common Data Format (CDF, see Appendix A) was devised for internal use by the RTG. This format represented a single gamma-ray spectrum as ASCII energy/count pairs preceded by an ASCII header. The ASCII representation of the data assured that it was compatible on any computing platform and this format is still in use.

In the mid 1990's it became apparent that instrument systems of greater complexity would demand a file format of larger capacity to support systems then on the drawing board, including networks of sensors collecting time series of gamma-ray spectra. These systems were in the planning stage and defined data structures were not available. It became apparent that a new storage format for nuclear measurements data would be needed and it would have to be flexible and extensible to accommodate the requirements of systems of the future. As part of an LDRD, we began to investigate what others were doing, especially in the high-energy physics community, to deal with the large volumes of data being generated.

Of particular interest was the very general Hierarchical Data Format (HDF), developed and maintained by the National Center for Supercomputing Applications (NCSA), that we ultimately used to develop the Radiation Physics Format (RPF). The HDF subroutine library provides users with the ability to customize a data file format based on standard calls to the HDF subroutine library. The RPF was developed and deployed on Sun and Hewlett-Packard workstations running their proprietary versions of UNIX.

An RPF spectral data file format consists of one or more data pages. A data page is composed of a header and its associated spectral data. It was implemented using the HDF *vset* paradigm which establishes a logical grouping of diverse, but related data items using only two basic types of storage elements: *vgroup*s and *vdata*s. *Vgroup*s contain only references to *vdata*s or other *vgroup*s. *Vdata* contain only data.

Reading, writing, and printing RPF data files is accomplished by calls to the RPFTools library, written by the author, that in turn relies on calls to the HDF subroutine library. The RPFTools library, developed in ANSI C, currently contains 18 functions for reading, writing, and printing RPF data files. Its dependencies are the HDF package, the standard C libraries, and clhtools, which is a collection of miscellaneous ANSI C utility functions, also written by the author.

In addition to the RPFTools, two stand-alone programs were written that illustrate the use of the library: **cdf2rpf** translates a CDF data file into an RPF file and **showrpf** allows a user to open and browse the contents of an RPF file.

It is important to summarize the fact that the RPFTools do not, of themselves, constitute a finished data format. They are tools for the creation of application-specific radiation physics data formats that can be structured to satisfy a user's requirements. This means that once such a structure is defined, some code must be written, using the RPFTools, to read, write, and print out the user's data. The RPFTools make the creation of that code a relatively simple task and the user can be assured that his or her data can be readily transferred intact and be ready to use on other computing platforms.

Upon reaching the stage of development described in this document, the RPFTools code suite was frozen at HDF version 4.1r3 pending the identification of a specific need. As a result, no final report was produced. This document is an unedited compendium of informal technical work papers produced by the author during various stages of the project. As a result some of the material is repetitive. Nevertheless, while informal, these papers give a view of the RPF at considerable technical depth.

RPF Spectral Data File Format

History

Neutron and gamma-ray spectrometry experiments can produce a large volume of data. At various times, these data need to be collected, annotated, collated, analyzed, transported, archived, and retrieved. In practice, these operations often occur on multiple platforms.

We needed a data file format that was flexible in the type and amount of data it could store, portable across different computer platforms, and was easily extensible to accommodate new data storage requirements. Finding an existing, well-developed framework that had freely distributed pre-compiled binaries for a wide variety of platroms; was open source for compilation on other platforms; had a large, diverse user base; and available user support was a bonus which helped to reduce the complexity and time required to implement a solution.

The Radiation Physics Format (RPF) spectral data file format is the result of designing a standard data file format for spectral data that meets these and other criteria. It is a self-describing binary file that is based upon the National Center for Supercomputing Applications (NCSA) Hierarchical Data format (HDF) paradigm.

Requirements

This file format must accommodate neutron (count) data, gamma-ray spectral (energy, flux pairs) data, and statistical data. These data can be either collected from or calculated for any number of detectors as either a single count/spectrum or a time series of counts/spectra.

In addition, the data files must also accommodate varying types and amounts of information describing the origin, history, and contents of the data. It is desired that this header information be readable by the user.

These data files must also be flexible/extensible, and portable across various platforms in order to avoid the problem of converting vast quantities of legacy data when modifying/adding information to the files, or changing platforms.

The implementation of these data files is based upon ANSI C using as few system-dependent extensions as possible in order to promote cross-platform compatibility. Current platforms under consideration are UNIX (HP, Sun), PC (DOS and Windows), and Mac (Mac-OS).

A pre-existing, widely used, noncommercial basis for the data files gives the advantages of not having to develop low-level routines from scratch, low cost availability, a potentially large user base from which to draw utilities and expertise, and freedom from dependence on an inflexible commercial product and the longevity of any particular company.

The HDF file format package available from NCSA provides a good basis for a solution to our problem. (The HDF WWW home page is found at *http://hdf.ncsa.uiuc.edu/.*) The NCSA anonymous ftp server is located at *ftp.ncsa.uiuc.edu.*) The following portion of this requirements section was excerpted from the beginning of the HDF FAQ and explains why the HDF framework was selected.

What is HDF?

HDF stands for Hierarchical Data Format. It is a multi-object file format for the transfer of graphical and numerical data between machines.

HDF is a versatile file format. It supports six different data models. Each data model defines a specific type of data and provides a convenient interface for reading, writing, and organizing a unique set of data elements.

HDF is a self-describing format, allowing an application to interpret the structure and contents of a file without any outside information.

HDF is a flexible file format. With HDF, you can group sets of related objects together and then access them as a group or as individual objects. There are pre-defined sets for raster images and floating point multidimensional arrays. User can also create their own grouping structures using an HDF feature called *vgroup*s.

HDF is an extensible file format. It can easily accommodate new data models, regardless of whether they are added by the HDF development team or by HDF users.

HDF is a portable file format. HDF files can be shared across platforms. An HDF file created on one computer, say a Cray supercomputer, can be read on another system, say IBM PC, without modification.

HDF is available in the public domain.

What is in the HDF library?

HDF currently supports six data structure types: 8-bit raster images, 24-bit raster images, color palettes, scientific data sets (multi-dimensional arrays), text entries and *vdata*s (binary tables).

The HDF library contains two parts: the base library and the multi-file library. HDF library functions can be called from C or FORTRAN user application programs.

The base library contains a general purpose interface and six application level interfaces, one for each data structure type. These application level interfaces are specifically designed to read, write and manipulate one type. The general purpose interface contains functions, such as file I/O, error handling, memory management and physical storage.

The multi-file part integrates netCDF model with HDF Scientific data sets, and supports simultaneous access to multiple file and multiple object. This part is referred to as netCDF/HDF library in the rest of this FAQ.

Conceptual Solution

Our previous attempt at implementing a spectral file format, the CDF file, provides a good starting point. A CDF file is an ASCII file which has a variable length and content header consisting of keyword labels followed by value (e.g. *nchan=4096*) delimited by the keywords *::beginheader::* and *::endheader::* followed by the spectral data. The spectral data could consist of up to five columns of information. More than one spectrum could be sequentially stored in one CDF file.

The advantages of this solution is that the header is variable length, variable content, and easy to read, write, and modify by an application and by the user with any text editor. This also leads to the disadvantages of non-standard keywords and that the user can easily corrupt the file.

The major disadvantage to CDF is the amount of storage space that the data can consume.

Specifications for the Actual Solution

The RPFTools software suite, and a couple of simple applications illustrating the suite, were written debugged and deployed on HP-UX, Solaris, and MacOS. A port to Windows was never completed. The suite was implemented using ANSI C standard code within the HDF version 4 framework. It is a binary variation on the CDF file solution in which there is a variable length, variable content header describing the HDF file content, and one or more sets of data with a variable length, variable content header describing the data.

The headers can be thought of as metadata since they are data about data and are implemented as HDF *vgroups.* The data file header consists of a variable number of vdata*s* that describe the data. The *vdata*'s name can be thought of as its description or variable name. The *vdata's* class is its category. And the *vdata*'s data are its values. For example, one header *vdata* entry might have a name of *location*, a class designation of *experiment_identification*, and a data value of *measurement lab.*

Data are represented by default as 16-bit integers or 64-bit floating point numbers depending upon their requirements unless specified otherwise by the user. Floating point numbers are automatically converted from the native format of the host machine to the standard HDF format of IEEE 32- or 64-bit floating point format as required. This is to promote cross-platform portability, a feature that is hindered by storing data using the native format option. Neutron data are stored as scientific data set (SDS) arrays of rank 3 to allow fields for time, neutron count, and detector. Gamma spectra are stored as SDS arrays of rank 4 to allow fields for time, energy, flux and detector. Calculated gamma spectra can have additional rank for relative error, absolute error, etc., if desired.

Editing an RPF Header

The use of the HDF model places a hurdle before the user who wants the advantages of an easily editable ASCII header. An HDF data file is binary. Tests performed on an annotated HDF data file show that the ASCII strings of the annotations are visible and comprehensible with listing utilities such as UNIX *more* and with text editors such as *emacs*. While RPF annotations are in order, they are spread throughout the RPF data file rather than being contiguous in any one place in the file. If you want to hand edit the string, you can. However, if you substitute another string with a different length, the whole HDF data file becomes unreadable. A similar situation has occurred many times in the past with CDF files. The HDF hurdle performs a useful service by encouraging the user to make header modifications with a more disciplined approach.

The disciplined approach is to create utilities for viewing, extracting, and replacing the header information. A viewing utility simply opens the RPF data file, extracts the header information, and displays it. A utility for extracting the header writes the header information to an ASCII file. This file can be hand edited by the user with his/her favorite text editor. A utility for replacing the header information will take this or any other valid ASCII header file plus an RPF data file and replace the header information of that RPF data file with that found in the ASCII header file.

This will provide the user with the flexibility and data integrity desired in a timely manner while minimizing demands on the implementer.

Keyword Integrity

There will be a standard set of keyword labels with specific definitions. This list will evolve with time and will be made available in documented form. Some header information updates in existing RPF data files may be required depending upon the scope and nature of the evolution of the RPF. This uniformity of labels and usage is essential for the successful integration of utilities and applications.

RPF Software Suite Overview

Introduction

The Radiation Physics Format (RPF) spectral data file format is an extensible, self-describing, binary data file format that can accommodate multiple sets of nuclear data within a single file and is portable across computing platforms. It is based upon the Hierarchical Data Format (HDF) software package developed and supported by the National Center for Supercomputing Applications (NCSA).

The RPFTools Software Suite comprises a variety of tools to initialize, read, write, view, and manipulate RPF data files. It contains source code, *include* files, linkable object modules of RPF-related and other miscellaneous functions, and stand-alone programs. It is written in ANSI standard C and links against the standard C I/O libraries and INCAS's HDF version 4.x libraries. Standalone programs exist to view RPF files and to create RPF files from existing CDF files. Tools for file conversion from other data formats can be made by writing programs using the appropriate functions in the RPFTools Software Suite.

Anatomy of an RPF file

An RPF spectral data file format consists of one or more data pages. A data page is composed of a header and its associated spectral data. It is implemented using the HDF *vset* paradigm which establishes a logical grouping of diverse, but related data items using only two basic types of storage elements: *vgroup* and *vdata*. *Vgroup*s contain only references to *vdatas* or other *vgroup*s. *Vdatas* contain only data.

The headers can be thought of as metadata since they are data about data. They comprise a variable number of keywords selected from a pre-established list of valid keywords and their values. The data page header is a *vset* that contains a variable number of *vgroups* and *vdatas* that describe the data. The *vdata*s name can be thought of as its description or variable name. The *vdata*'s class is its category. And the *vdata*s data are its values. For example, one header *vdata* entry might have a name of *location*, a class designation of *experiment_identification*, and a data value of *measurement lab.*

Data are represented by default as 32-bit integers or 64-bit floating point numbers depending upon their requirements unless specified otherwise by the user. Floating point numbers will automatically be converted by the underlying HDF package from the native format of the host machine to the standard HDF format of IEEE 32- or 64-bit floating-point format as required. This is to promote cross-platform portability, a feature that is hindered by storing data using the native format option.

Neutron data are stored as Scientific Data Set (SDS) arrays of rank 3 to allow fields for time, neutron count, and detector. Gamma spectra converted from the locally developed Common Data Format (CDF is an early Radiation Technology group data file format) data files are stored as SDS arrays of rank 3 to allow fields for bin edge energy, bin center energy, and flux. Rank can be increased to allow fields for time and detector. Calculated gamma spectra can have additional rank for relative error, absolute error, etc., if desired.

RPF data can originate as instrumental data or computational simulations of experiments from single detectors, detector arrays, and networks of disparate detectors. Total count data can be stored as a time series or as multichannel scaling. Spectral data can currently be stored as pulse-height spectra. Time-stamped list mode storage for spectral data is planned for a later version. Data can be mixed within a single RPF file. Time series of spectra or gross count data from multiple detectors can be stored within a single RPF file. This would allow, for example, disparate measurements from a series of related experiments or from arrays of detectors from a network. RPF is an important new contribution to our core competency and has broad applicability.

Platforms

The RPFTools Software Suite was initially developed on an HP 9000 model 735 running HP-UX v 9.01 using the HDF version 4.0r2 release that was precompiled for HPUX v 9.03. The current version, v2.2, was developed on the same HP platform using the HDF version 4.1b1 release that was precompiled for HPUX v 9.03.

The RPFTools Software Suite has also been ported to a Sun SPARCstation 10/512 running Solaris 2.5.1 using the HDF version 4.0r2 release that was precompiled for Solaris 2.4. The current version is v1.0b2.

It has been demonstrated that an RPF file created on the HP or Sun can be transferred to the Sun or HP, respectively, and viewed.

RPFTools library

The RPFTools library, rpftools.o, is generated from the source code module, rpfools.c, written in ANSI C. It currently contains 18 functions for reading, writing, and printing RPF data files. Its dependencies are the HDF package, the standard C libraries, and clhtools, which is a collection of miscellaneous ANSI C utility functions.

Function name	*Function description*
init_rpf_header	Initializes the space to hold the RPF header information and returns a pointer to the new RPF header data structure.
reset_rpf_header	Resets the RPF header values to initialized values.
free_rpf_header	Frees the space used by the RPF header.
init_rpf_header_status	Initializes the space to hold the RPF header status and returns a pointer to the RPF header status data structure
add_rpf_comment	Adds a string to the RPF header comment field.
parse_cdf_header	Parses the header of a local-CDF file and fills in the corresponding variables in a RPF header.
print_current_rpf_header	Prints the portion of the RPF header that is currently valid to stdout.
print_full_rpf_header	Prints the full RPF header to stdout.
read_open_rpf_file	Opens an RPF file for reading.
write_open_rpf_file	Opens an RPF file for writing.
close_rpf_file	Closes an RPF file.
read_rpf_header	Reads an RPF header from an RPF data file.

write_rpf_header	Writes an RPF header into an RPF data file.
read_rpf_data	Reads the RPF data from an RPF data file and returns pointer to the array of RPF data values.
write_rpf_data	Writes the RPF data into an RPF data file and returns the reference number of the RPF data scientific data set (SDS).
print_rpf_data	Prints the RPF data to stdout.
read_rpf_page	Reads an RPF page from an RPF file.
write_rpf_page	Writes an RPF page to an RPF file.

Stand-alone applications

There are currently two standalone RPF applications in the RPFTools Software Suite. However, showrpf requires some additional debugging on the Sun.

Program name	***Program description***
cdf2rpf	Translates a CDF data file into an RPF file.
showrpf	Generates user-controlled display an RPF file to stdout.

RPFTools Reference Manual

Introduction

The Radiation Physics Format (RPF) spectral data file format is an extensible, self-describing, binary data file format which is portable across computing platforms. It is based upon the Hierarchical Data Format (HDF) software developed by the National Center for Supercomputing Applications (NCSA).

This document describes the Radiation Physics Format (RPF) spectral data file format and the software tools used to manipulate the RPF data files.

Information Pages

An RPF data file consists of one or more pages of information. A page consists of a header describing the spectral data plus the associated spectral data.

Each page of information is stored as an HDF *vgroup* in the RPF file. The *vgroup* class is *rpf_page*. The page name is generated by concatenating the string *page_*, the page index and the string "_" before it is stored as the *vgroup* name. There is a header *vgroup* and a data *vgroup* associated with each page *vgroup.* The *vgroup* classes of these are *rpf_page_header* and *rpf_page_data,* respectively.

The header *vgroup* consists of a variable number of keyword *vdatas* that encapsulate the header information. The name of the keyword *vdata* is the variable name in the corresponding RPF header data structure. The keyword *vdata* class is the name of the keyword category designation for that keyword inside the RPF *header_type* data structure. The keyword data fields are defined with the appropriate data type and named the same as the keyword *vdata.* Only those keywords which contain valid data as indicated by their corresponding *rpfheader_status* fields being set to ON will be written to the RPF file when the header is stored.

The data *vgroup* consists of a multidimensional data block which is stored as a scientific data set (SDS). The dataset name is generated by concatenating the string *data_*, the data index and the string "_." The *data label* attribute is set to *gamma_data*. The first three dimensions are labeled *bins*, *units*, and *boxes*, respectively to designate, single spectra, temporal series of spectra, and spectra gathered from array of detectors. Single spectra are normally concerned with energy bins. Temporal spectra are concerned with spectra taken within slices of time. A *box* of spectra would be those collected from an array of detectors. Alternatively, one may think of the second dimension as adding spectra and the third dimension as adding detectors.

RPF Software Tools

The RPFTools software suite comprises of a variety of tools to initialize, read, write, view, and manipulate RPF data files. It contains source code, *include* files, linkable object modules of RPF related and other miscellaneous functions, and standalone programs. It is written in ANSI standard C and links against the standard C I/O libraries and NCSA's HDF version 4.3 libraries.

The *include* file, *rpftools.h,* defines a data structure to contain the RPF header information, plus a data structure to hold an indicator variable for each of the keyword fields, in addition to the function declarations for each of the functions defined in *rpftools.c.* For a more detailed description of the keywords contained in the RPF header, see the document, *RPF keyword data dictionary.*

The RPFTools library, *rpftools.o,* is generated from the source code module, *rpftools.c.* It contains various ANSI C functions for reading, writing, and printing RPF data files. Its dependencies include the HDF *include* files (HDF *inc* directory) and libraries (HDF *bin* and *lib* directories), and the standard C libraries (*include* files and object modules). Non-HDF *include* files required at compilation time are *rpftools.h,* and *clhtools.h.* Non-HDF object modules required at linking time are *rpftools.o,* and *clhtools.o.* The two files, *clhtools.h* and *clhtools.o,* are the *include* file and object module, respectively, for a collection of miscellaneous ANSI C utility functions used by RPFTools.

Data Structure Definitions

Example Usage. Use *read_rpf_page* to read a page of information from an RPF file. Use *write_rpf_page* to write a page of information to an RPF file.

Reference Section Overview

This section of the RPFTools Reference Manual contains a listing of every function contained in RPFTools.c. The functions are ordered alphabetically according to their C function name. Each function is described in the following form:

Function name

Return_type function_name (type paramaeter1, type parameter2, … type parameter2)

Parameter1	Definition of the first parameter
Parameter2	Definition of the second parameter
…	
ParameterN	Definition of the Nth parameter

Purpose	Section containing a short description of the function.
Return value	Section describing the return value, if any, and error conditions.
Description	This section describes the function and any special circumstances surrounding the use of this function. It identifies any functions that must precede it and describes any known complications related to the use of it.
Example	This optional section consists of one or more examples of the use of the function. The examples are not complete programs and are intended for illustrative purposes only. For complete programs and further descriptions of the function, refer to the RPFTools User's Guide.

Functions

add_rpf_comment

void add_rpf_comment (header_type *rpfheader, header_status_type *rpfheader_status, char *t)

rpfheader	OUT:The space to store the RPF header information
rpfheader_status	OUT:The space to store the RPF header status information
t	IN:The string to add to the comment field
Purpose	Adds a string to the RPF header comment field.
Return Value	None.
Description	This function adds a string to the header comment field. If there is already an existing comment, a newline ("\n") is added before the new comment is appended. The resulting string is null terminated ("\0") and the rpfheader status field, rpfheader_status->got_valid_one, is set to ON before the function returns.

close_rpf_file

void close_rpf_file (int32 *file_id, int32 sd_id, int32 sds_id, int32 vgroup_id)

file_id	IN:The RPF file id
sd_id	IN:The RPF data SD id
sds_id	IN:The RPF data SDS id
vgroup_id	IN:The RPF data vgroup id
Purpose	Terminates access to the SD interface and the V interface, and then closes an RPF file.
Return Value	None.
Description	This function terminates access to the SD interface designated by the sd_id and the sds_id, terminates access to the V interface designated by the vgroup_id, and then closes an RPF file designated by the file_id.

free_rpf_header

void free_rpf_header (header_type *rpfheader)

rpfheader	IN: A pointer to the RPF data structure to destroy.
Purpose	Frees the space used by the RPF header.
Return Value	None.
Description	This function frees the memory space previously allocated to hold the full RPF header information. Use this destructor function to avoid memory leaks due to incomplete release of header storage space.

get_line

int get_line (char *s, FILE *fp)

s	OUT:The string of input data
fp	IN:The pointer to the pre-opened file to be read
Purpose	Get a single valid line of input data while ignoring comment lines.
Return Value	It returns a 0 if it finds the EOF at the beginning of a line, a -1 if it finds a # at the beginning of a line (comment line), and a 1 otherwise.
Description	This function grabs a single valid line of input data while ignoring comment lines that are indicated by a leading #. The original version was written for the UNIX operating system and obtained from Mark Wagner. During the most recent port to the Macintosh platform, it was rewritten. It now grabs a line as delimited by a newline (\n), a carriage return (\r) or an end of file (EOF).

have_rpf_file

void have_rpf_file (char *rpffname, int *eflag)

rpffname	IN:The name of the RPF file
eflag	OUT:An error flag, initialized to 0.
Purpose	Determines whether a file is an RPF file.
Return Value	None.
Description	This function determines whether the file specified by rpffname is indeed an RPF file. Eflag is set to 1 if the file not found. It is set to 2 if the file that is found is not an RPF file.

init_cdf_data

specdat_type *init_cdf_data (int tsize)

tsize	IN:the initial number of bins in the data arrays
Purpose	Initialize the space to hold the CDF data.
Return Value	A pointer to the initialized CDF data structure.

Description This function creates and then initializes the space to hold the CDF data.

If tsize is less than or equal to zero, tsize defaults to 42. The data arrays, e_center, flux, d1, d2, and d3 are malloc-ed to a length of tsize and each value is set to 0.0. The e_edge data array is malloc-ed to a dize of tsize + 1 to accommodate the bin edges and then each value is set to 0.0.

The parameters nbins and xstart are set to 0. The parameter asize is set to tsize.

init_rpf_header

header_type *init_rpf_header (void)

Purpose Initializes the space to hold the RPF header information.

Return Value Returns a pointer to the newly created and initialized RPF header data structure.

Description This function initializes the space to hold the full RPF header information. Character string variables are initialized to hold one null character ("\0") to show that they start off as empty strings. Integer variables are set to 0. Double variables are set to 0.0. Pointers to doubles are malloc-ed to a length of 1 and the value of the first array space is set 0.0. See the RPF Keyword Data Dictionary for a complete description of the RPF header fields.

init_rpf_header_status

header_status_type *init_rpf_header (void)

Purpose Initializes the space to hold the RPF header status information.

Return Value Returns a pointer to the RPF header status data structure.

Description This function initializes the space to hold the status for all the fields in the full RPF header and sets the value of all status fields to OFF (don't have). As the RPF header fields filled their corresponding status field values will be set to ON. OFF and ON are defined in RPFTools.h to be 0 and !OFF, respectively.

parse_cdf_header

void parse_cdf_header (char *fname, header_type *rpfheader, header_status_type *rpfheader_status, int verbose, int *eflag)

fname	IN:name of the CDF file.
rpfheader	OUT:A pointer to the RPF header information.
rpfheader_status	OUT:A pointer to the RPF header status information.
verbose	IN:A flag to control the amount of output.
eflag	OUT:An error flag, initialized to 0.
Purpose	Parses the header of a CDF file and fills the corresponding variales in a RPF header.
Return Value	None.
Description	This function opens an ASCII CDF file, parses its header line by line, places whatever information it can into a RPF header structure, and sets the corresponding variables in the RPF header status structure to ON.

If the verbose flag is set to ON, when an RPF keyword is matched, it and its corresponding data value are printed to stdout. Also, when no RPF keyword is matched in the line of CDF header the message did not match rpf keyword:' followed by the CDF line is printed to stdout.

If the number_of_bins keyword value is redefined, a warning will be printed to stdout and eflag will be changed from its initial value of 0 to 1. This can happen when the CDF keywords *nchan* and/or *Channels=* occurs more than once within the same CDF header section.

Note that the CDF file referred to here is the Common Data Format developed at LLNL and should not be confused with netCDF developed by UCAR.

This function covers only a small number of the possible keywords that can be found in a CDF file. Please see *An Introduction to the Common Data Format (CDF)* for further information for additional information.

print_current_rpf_header

void print_current_rpf_header (header_type *rpfheader, header_type *rpfheader_status)

rpfheader	IN:A pointer to the space which stores the RPF header information
rpfheader_status	IN:A pointer to the space which stores the RPF header status information

Purpose	Prints the portion of the RPF header that is currently valid.
Return Value	None.
Description	This function prints the fields of the RPF header data structure whose corresponding status variables are currently set to ON to stdout.

print_full_rpf_header

void print_full_rpf_header (header_type *rpfheader, header_type *rpfheader_status)

rpfheader	IN:A pointer to the space which stores the RPF header information
rpfheader_status	IN:A pointer to the space which stores the RPF header status information
Purpose	Prints the full RPF header.
Return Value	None.
Description	This function prints the full RPF header data structure to stdout.

print_rpf_data

void print_rpf_data (double *rpfdata, int32 rank, int32 *dims, int npl)

rpfdata	IN:The buffer containing the RPF data
rank	IN:The rank of the data block to be printed
dims	IN:Array specifying the size of each dimension
npl	IN:Number of data values to print per line
Purpose	Prints the RPF data.
Return Value	None.

Description

This function prints the RPF data to stdout. The rank and dimension size of the data block is followed by the data itself. Currently, this function is set up for a rank of 2.

An example output looks as follows:

```
rank: 2
index: 0   dims[0]: 4
index: 1   dims[1]: 3
 [0][0]: 2   [1][0]: 3   [2][0]: 24
 [0][1]: 4   [1][1]: 5   [2][1]: 26
```

[0][2]: 6   [1][2]: 7   [2][2]: 28

[0][3]: 8   [1][3]: 0   [2][3]: 0

readcdf

void readcdf (char *fname, int *nbin, EF_TYPE **dat1, EF_TYPE **dat2, EF_TYPE **dat3, EF_TYPE **dat4, EF_TYPE **dat5, int max_size, int *eflag)

fname	IN:The name of the CDF file to be opened, read, and then closed
nbin	OUT:The number of energy bins in the data matrix
dat1	OUT:A pointer to the energy bin data array
dat2	OUT:A pointer to the flux data array
dat3	OUT:A pointer to the relative error array
dat4	OUT:A pointer to the absolute error data array
dat5	OUT:A pointer to the original flux data array
max_size	IN:The maximum allowed size of the data array
eflag	OUT:An error flag, initialized to 0.

Purpose Read an ASCII CDF file containing up to five data values per line into separate double arrays.

Return Value None.

Description This function reads an ASCII CDF file containing from two to five data values per line into separate EF_TYPE arrays. The read continues is until an EOF is hit. Comments as indicated by a leading # and blank lines are ignored.

The data arrays are resized as needed up to a maximum size of max_size elements. If the data to be returned occupies less than 75% of the malloc-ed space, the arrays are realloc-ed to use only the required space.

The parameter eflag is incremented each time it encounters a line of data with only one number. It is set to -1 if the *::ENDHEADER::* or *::endheader::* keyword is not found. It is set to –2 if there were no valid lines of data found.

read_open_rpf_file

void read_open_rpf_file (char *rpffname, int32 *file_id, int32 sd_id, int *eflag)

rpffname	IN:The name of the RPF file

file_id	OUT:The RPF file id returned by HDF function Hopen
sd_id	OUT:The RPF data SD id returned by HDF function SDstart
eflag	OUT:An error flag, initialized to 0.
Purpose	Opens an RPF file for reading.
Return Value	None.
Description	This function opens an RPF file for reading, initializes the SD interface, and initializes the RPF file for subsequent vgroup access. The file_id and the sd_id are returned in the parameter list. Eflag is set to 1 if the file not found. It is set to 2 if the file that is found is not an RPF file.

read_rpf_data

double *read_rpf_data (int32 vgroup_id, int32 sd_id, int32 *rank, int32 *dims, int *eflag)

vgroup_id	IN:The RPF vgroup id
sd_id	IN:The RPF data SD id
rank	OUT:The rank of the data to be read
dims	OUT:Array specifying the size of each dimension
eflag	OUT:An error flag, initialized to 0.
Purpose	Reads the RPF data from an RPF data file.
Return Value	Pointer to the array of RPF data values.
Description	This function reads the RPF data section from a previously opened rpf file and returns a pointer to the buffer containing the RPF data. The rank and the dimension of the data is returned in the parameter list. The space for dims needs to be allocated before entering this function. A safe way would be to define it as dims[MAX_VAR_DIMS]. MAX_VAR_DIMS is defined as 32 in the HDF *include* file hlimits.h. Eflag is incremented by 1 for each read error that occurs from a call to SDreaddata.

read_rpf_header

void read_rpf_header (int32 file_id, int32 vgroup_id, header_type *rpfheader, header_status_type *rpfheader_status, int *eflag)

file_id	IN:The RPF file id previously returned by HDF function Hopen.

vgroup_id	IN: The id of the header vgroup to be read.
rpfheader	OUT:The RPF header information.
rpfheader_status	OUT:The RPF header status information.
eflag	OUT:An error flag, initialized to 0.
Purpose	Reads an RPF header from an RPF data file.
Return Value	None.
Description	This function reads the RPF header vgroup from a previously opened rpf file. It fills the RPF header data structure and sets the corresponding indicator variables in the header_status structure to "ON".
	Eflag is incremented by 1 for each read error. It is incremented by 1000 for each time the vdata class and name does not match those for an RPF keyword.

read_rpf_page

void read_rpf_page (char *rpffname, int pageindex, header_type **rpfheader, header_status_type **rpfheader_status, double **rpfdata, int32 *rank, int32 *dims, int *eflag)

rpffname	IN:The name of the RPF file
pageindex	IN:The index of the RPF page to read.
rpfheader	OUT: A pointer to the RPF header structure
rpfheader_status	OUT: A pointer to the RPF header status structure
rpfdata	OUT:A pointer to the buffer containing the RPF data
rank	OUT:The rank of the data to be read.
dims	OUT:Array specifying the size of each dimension in the data block.
eflag	OUT:An error flag, initialized to 0.
Purpose	Reads a RPF page.
Return Value	None
Description	This function opens a RPF file and reads a page of RPF data.
	The space for dims, start, and edges need to be allocated before entering this function. A safe way would be to define dims as dims[MAX_VAR_DIMS]. MAX_VAR_DIMS is defined as 32 in the HDF *include* file hlimits.h.
	Eflag is initialized to 0. It is set to –1 if the RPF page designated by pagename that is created from the pagein-

dex is not found and the function returns from there. Eflag is set to –2 if there were any errors during the reading of the header vgroup. Eflag is set to –3 if there were any errors during the reading of the data vgroup. It is set to –4 if errors occurred during the reading of both the header and data vgroups.

reset_rpf_header

void reset_rpf_header (header_type *rpfheader)

rpfheader	IN: The RPF data structure to reset.
Purpose	Resets the RPF header values to initialized values.
Return Value	None.
Description	This function resets the values for the full RPF header. Character string variables are initialized to just hold one \0 to show that they start off as empty strings. Integer variables are set to 0. Double variables will be set to 0.0. Pointers to doubles are initialized to a length of 1 and the value of the first array space is set 0.0. See the RPF Keyword Data Dictionary for a complete description of the RPF header fields.

strchop

void strchop (char *instring, char *outstring)

instring	IN:The original string
outst	OUT: The original string truncated to remove trailing white spaces
Purpose	Remove the trailing white spaces in a string.
Return Value	None.
Description	This function removes the trailing white spaces in a string. The resulting string is null terminated (‘\0’).

write_open_rpf_file

void write_open_rpf_file (char *rpffname, int32 *file_id, int32 sd_id, int *eflag)

rpffname	IN:The name of the RPF file
file_id	OUT:The RPF file id returned by HDF function Hopen
sd_id	OUT:The RPF data SD id returned by HDF function SDstart
eflag	OUT:An error flag, initialized to 0.
Purpose	Opens an RPF file for writing.

Return Value None.

Description This function opens an RPF file for writing. The SD interface is then initialized and the file is prepared for vgroup access.

Eflag is set to 1 if the file not found. A new RPF file is then created with read/write access. Eflag is set to 2 if the file that is found is not an RPF file. The function then returns.

write_rpf_data

int32 write_rpf_data (int32 file_id, int32 sd_id, char *datasetname, double *rpfdata, int32 rank, int32 *dims, int32 *start, int32 *edges, int *eflag)

file_id	IN:The RPF file id returned by HDF function Hopen
sd_id	IN:The RPF data SD id returned by HDF function SDstart
datasetname	IN:The name of the RPF data vgroup.
rpfdata	IN:The buffer containing the RPF data.
rank	IN:The rank of the data to be written.
dims	IN:Array specifying the size of dimension.
start	IN:Array specifying the starting location.
edges	IN:Array specifying the number of values to be written along each dimension.
eflag	OUT:An error flag, initialized to 0.

Purpose Writes the RPF data into an RPF data file.

Return Value The reference number of the RPF data SDS vgroup_id.

Description This function opens a RPF file, and writes the RPF data as an HDF scientific data set (SDS). It is up to the user to create the rpfdata block to pass to this function.

For example, Type 2 CDF data storing a single gamma ray spectrum can be written as a rank 2 data matrix in which the 0th dimension, dims[0], is the number of bin edges (equivalent to the number of bins + 1), and the 1st dimension, dims[1], is equal to 3. This would facilitate the storage of energy bin edge values, energy bin center values and bin fluxes.

Eflag is set to the number of errors that occurred during the writing of the data.

write_rpf_header

int32 write_rpf_header (int32 file_id, char *headername, header_type*rpfheader, header_status_type *rpfheader_status, int *eflag)

file_id	IN:The RPF file id previously returned by HDF function Hopen
headername	IN:The name of the RPF header vgroup.
rpfheader	IN:The space which stores the RPF header information.
rpfheader_status	IN:The space which stores the RPF header status information.
eflag	OUT:An error flag, initialized to 0.
Purpose	Writes an RPF header into an RPF data file.
Return Value	The vgroup_id of the header vgroup that is written
Description	This function writes the RPF header information into a vgroup of a previously opened rpf file designated by its file_id. Only those keywords whose corresponding rpfheader_status fields are set to ON will be written. Eflag is set to the number of errors that occurred during the writing of the header data using VSwrite.

write_rpf_page

void write_rpf_page (char *rpffname, int pageindex, header_type **rpfheader, header_status_type **rpfheader_status, double **rpfdata, int32 *rank, int32 *dims, int32 *start, int32 *edges, int *eflag)

rpffname	IN:The name of the RPF file.
pageindex	IN:The index of the RPF page to return.
Rpfheader	OUT: A pointer to the pointer to the RPF header structure.
rpfheader_status	OUT: A pointer to the pointer to the RPF header status structure.
rpfdata	OUT:A pointer to the buffer containing the RPF data.
rank	OUT:The rank of the data to be written.
dims	OUT:Array specifying the size of dimension.
start	OUT:Array specifying the starting location.
edges	OUT:Array specifying the number of values to be written along each dimension.
eflag	OUT:An error flag, initialized to 0.
Purpose	Writes a RPF page.

Return Value None.

Description This function opens a RPF file and writes a page of RPF data.

Eflag is initialized to 0. It is set to –1 if the file designated by rpffname is found not to be an RPF file and the function returns from there. Eflag is set to –2 if there were any errors during the writing of the header vgroup. Eflag is set to –3 if there were any errors during the writing of the data vgroup. It is set to –4 if errors occurred during the writing of both the header and data vgroups.

Data Dictionary: Header Keywords for the Radiation Physics Format

Description

It is imperative for the portability of data between our applications to have a standard set of keywords in the header information section(s) of our RPF files. This section contains the data dictionary for the keywords that are used to describe the fields in our RPF files. The initial list of keywords was declared by fiat by the RPF file designer, Cheri Ham (ham1@llnl.gov) after receiving input from users Tom Gosnell (gosnell1@llnl.gov), Zach Koenig (koenig@llnl.gov), Bert Pohl (pohl1@llnl.gov), Dave Knapp (knapp2@llnl.gov), Jim Wolford (wolford@llnl.gov), and Alexis Schach von Wittenau (schachvonwittenau@llnl.gov).

In order for the RPF spectral file header format and its associated software to be flexible and yet maintain their extensibility with minimal backtracking and reworking, the definitions of the header keywords and their uses must be strictly controlled. It is expected that the list of header keywords will evolve with use over time. Procedures to add, modify, delete, and exclude header keywords are described below. These procedures, although draconian in appearance, must be strictly followed until or unless they are formally modified. Maintaining a tight rein on the keywords from the beginning will lessen the chaos that keyword changes will cause in the future.

Remember, not all keywords will appear in each RPF header. Only those keywords relevant to each particular header will be included in the RPF file.

Adding/modifying keywords

Changes and additions may be made to the accepted keyword list according to the following 6-step procedure.

Step 1: Submit written request for data dictionary modification to the person in charge of the keyword data dictionary. Initially and currently the keyword Czar will be the RPF file designer, Cheri Ham. For modifications of existing keywords, this request should include the a list of the current keyword(s) to be modified, a list of the corresponding keyword(s) that the requester would like to be used, and a justification for each of the requested changes. To incorporate new keywords, the request should include a list of the new keyword(s) to be added, a complete definition of each of the new keywords, and justification for each of the requested new keywords.

Step 2: The keyword Czar will submit the written request to the appropriate subset of the local community of RPF file users for feedback. They will have 7 calendar days to provide oral and/or written feedback. Initially and currently, the members of the keyword review committee will be Tom Gosnell (gosnell1@llnl.gov), Zach Koenig (koenig@llnl.gov), Bert Pohl (pohl1@llnl.gov), Dave Knapp (knapp2@llnl.gov), Jim Wolford (wolford@llnl.gov), and Alexis Schach von Wittenau (schachvonwittenau@llnl.gov).

Step 3: The keyword Czar will consider the initial request and all the feedback, resolve any and all conflicts, and render a ruling upon the request, plus generate the required update to the RPF spectral file accepted keyword data dictionary.

Step 4. A detailed written ruling on the request to add/modify keywords will be disseminated to the local RPF file user community. This will include information on the original request, the feedback received, and the ruling rendered, plus any other relevant information such as the potential keyword data dictionary entry. The local user community will have 7 calendar days to submit a written appeal to the keyword Czar.

Step 5. They keyword Czar will resolve any appeals by repeating the described process as required.

Step 6. They keyword Czar will make a detailed entry into the keyword data dictionary logbook and update the keyword data dictionary as required.

Deleting/excluding keywords

There will be occasions that certain keywords and the information they represent will be deleted or excluded from the accepted list of keywords. The following describes the procedure to delete or exclude a keyword from the accepted header keyword list.

Step 1: Submit written request for data dictionary modification to the person in charge of the keyword data dictionary. Initially and currently the keyword Czar will be the RPF file designer, Cheri Ham. This request should include the a list of the current keyword(s) to be deleted or excluded and a justification for each of the requested changes.

Step 2: The keyword Czar will submit the written request to the appropriate subset of the local community of RPF file users for feedback. They will have 7 calendar days to provide oral and/or written feedback. Initially and currently, the members of the keyword review committee will be Tom Gosnell (gosnell1@llnl.gov), Zach Koenig (koenig@llnl.gov), Bert Pohl (pohl1@llnl.gov), Dave Knapp (knapp2@llnl.gov), Jim Wolford (wolford@llnl.gov), and Alexis Schach von Wittenau (schachvonwittenau@llnl.gov).

Step 3: The keyword Czar will consider the initial request and all the feedback, resolve any and all conflicts, and render a ruling upon the request, plus generate the required update to the RPF spectral file rejected keyword data dictionary.

Step 4. A detailed written ruling on the request to delete/exclude keywords will be disseminated to the local RPF file user community. This will include information on the original request, the feedback received, and the ruling rendered, plus any other relevant information. The local user community will have 7 calendar days to submit a written appeal to the keyword Czar.

Step 5. They keyword Czar will resolve any appeals by repeating the described process as required.

Step 6. They keyword Czar will make a detailed entry into the keyword data dictionary logbook and update the keyword data dictionary as required.

Modifying the keyword change procedure

It its allowed that the procedures for changing the header keyword list might require modification at some point in time.

Step 1: Submit written request for procedure modification to the person in charge of the keyword data dictionary. Initially and currently the keyword Czar will be the RPF file designer, Cheri Ham. The requester should include a description of and a justification for each of the requested changes, additions, and/or deletions.

Step 2: The keyword Czar will consider the request, and if deemed necessary will submit the written request to the appropriate subset of the local community of RPF file users for feedback. They will have 7 calendar days to provide oral and/or written feedback. Initially and currently, the members of the keyword review committee will be Tom Gosnell (gosnell1@llnl.gov), Zach Koenig (koenig@llnl.gov), Bert Pohl (pohl1@llnl.gov), Dave Knapp (knapp2@llnl.gov), Jim Wolford (wolford@llnl.gov), and Alexis Schach von Wittenau (schachvonwittenau@llnl.gov).

Step 3: The keyword Czar will consider the initial request and all the feedback, resolve any and all conflicts, and render a ruling upon the request, plus generate the required update to the Modifying the keyword change procedure.

Step 4. A detailed written ruling on the request to add/modify keywords will be disseminated to the local RPF file user community. This will include information on the original request, the feedback received, and the ruling rendered, plus any other relevant information. The local user community will have 7 calendar days to submit a written appeal to the keyword Czar.

Step 5. They keyword Czar will resolve any appeals by repeating the described process as required.

Step 6. They keyword Czar will make a detailed entry into the keyword data dictionary logbook and update the keyword change procedure as required.

RPF spectral file accepted keyword data dictionary

This data dictionary contains a list of the accepted HDF spectral file keywords, their defined data type, an example, and a detailed description of the information that they contain. This list is ordered to group keywords by functionality as opposed to alphabetically.

File creation keywords

The following three keywords preserve information on the initial creation of the HDF spectral data file.

date_file_written	char *	Ex: mm-dd-yyyy e.g., 03-14-1995. This ASCII string contains the original date that this file was written.
time_file_written	char *	Ex: hh:mm:ss XXX e.g., 14:12:00 PST. This ASCII string contains the original time that this file was written.
original_filename	char *	E.g., \data\U071_414.CNF. This ASCII string contains the original file name used when this file was written.

Experiment identification keywords

The following keywords contain basic information to identify the experiment.

location	char *	This ASCII string contains the location where the experiment was performed.
experimenters_names	char *	This ASCII string contains the names of the people involved in conducting the experiment.
experiment_id	char *	This ASCII string contains the experiment identification label.
run_number	int32	This is the run number which resulted in the data attached to this header.
number_of_channels	int32	This contains the number of channels in the data block associated with this header.

Hardware Identification Keywords

These keywords identify the experimental hardware and their parameters. The use of high_voltage, fine_gain, coarse_gain, and shaping_time are dependent upon the type of instrument used.

instrument_type	char *	This ASCII string describes the type of instrument used in the experiment.
instrument_id	char *	This ASCII string contains the instrument serial number or other identification.
high_voltage	real64	This describes the high voltage adjustment used.
fine_gain	real64	This describes the fine gain adjustment used.
coarse_gain	real64	This describes the coarse gain adjustment used.
shaping_time	real64	This describes the shaping time used.
detector_type	char *	This ASCII string describes the type of detector used in the experiment.
detector_id	char *	This ASCII string contains the detector serial number or other identification.

Experiment Time Keywords

The following keywords contain timing and location information for the experiment.

count_clock_time	char *	Ex: hh:mm:ss XXX e.g., 14:12:00 PST. This ASCII string contains the clock time when the experiment was started stored in ANSI standard format.
count_start_date	char *	Ex: mm-dd-yyy e.g., 03-14-1995. This ASCII string contains the date when the experiment’s data collection was started stored in ANSI standard format.
count_start_time	char *	Ex: hh:mm:ss XXX e.g., 14:12:00 PST. This ASCII string contains the time when the experiment’s data collection was started stored in ANSI standard format.
count_live_time	real64	This contains the live time of the experiment measured in decimal seconds.
count_dead_time	real64	This contains the dead time of the experiment measured in decimal seconds.

Energy Calibration Keywords

These keywords describe various energy calibration parameters.

energy_calibration_type	char *	This describes the type of energy calibration used. Valid options are none, normal, polynomial, and binned. None means that no calibration is available. Normal uses the keywords intercept, slope, and quadratic to calculate the energy calibration via the formula energy value = [quadratic * bin_number *bin_number] + [slope * bin_number] + [offset]. Polynomial uses the number_of_energy_coeffs coefficients stored in energy_calibration_coeffs to calculate the energy calibration using a higher order polynomial equation. The intercept is stored in energy_calibration_coeffs[0]. The slope is stored in energy_calibration_coeffs[1]. The quadratic term is stored in energy_calibration_coeffs[2], and so on. The spectrum option stores the number_of_energy_bin_edges bin edges in the energy_calibration_spectrum array.
intercept	real64	This is the intercept for the normal mode of energy calibration.
slope	real64	This is the slope for the normal mode of energy calibration.
quad	real64	This is the quadratic factor for the normal mode of energy calibration.
number_of_energy_coeffs	int16	This is the number of energy coefficients for the polynomial calibration option.
energy_calibration_coeffs	real64 array	This contains the actual coefficients for the polynomial calibration option. The intercept is stored in energy_calibration_coeffs[0]. The slope is stored in energy_calibration_coeffs[1]. The quadratic term is stored in energy_calibration_coeffs[2], and so on.
number_of_energy_bin_edges	int16	This is the number of energy bin edges required for the energy calibration. This value should equal [number_of_channels + 1].

energy_calibration_spectrum	real64 array	This contains the energy bin edges for the data associated with this header. energy_calibration_spectrum[0] contains the leftmost bin edge. The remaining [number_of_energy_bin_edges - 1] bin edges are the energies of the right bin edges.
energy_calibration_filename	char *	This ASCII string contains the name of the file from which energy calibration information was obtained for the data associated with this header.

Efficiency calibration keywords

These keywords describe various peak efficiency calibration parameters.

efficiency_calibration_type	char *	This describes the type of efficiency calibration used. Valid options are none, parameter, and spectrum. None means that no calibration is available. Parameter uses the number_of_efficiency_coeffs parameters stored in efficiency_calibration_params to store the efficiency calibration information. The spectrum option stores the efficiency calibration information in the efficiency_calibration_spectrum array.
efficiency_calibration_filename	char *	This ASCII string contains the name of the file from which energy calibration information was obtained for the data associated with this header

Geometry keywords

These keywords describe the geometry of the experiment.

source_detector_distance	real64	This is the closest approach [a.k.a. fact-to-face] distance from the source to the detector in meters. It is assumed to be the closest approach distance unless the **geometry_description** field describes otherwise.
geometry_description	char *	This describes relevant experimental geometry information.
%_solid_angle	real64	This is the % solid angle.

Material description keywords

These keywords describe the material(s) used in the experiment and are grouped according to source, absorber, collimator, and shield parameters that are described in more detail below.

Source Description Keywords

These keywords describe the source used in the experiment.

source_description	char *	This describes the source used in the experiment.
source_id	char *	This is the serial number or other identifier for the source used in the experiment.
source_material	char *	This is the index of the source material.
declared_enrichment	real64	This is the declared enrichment value.
declared_enrichment_error	real64	This is the error associated with the declared enrichment value.
measured_enrichment	real64	This is the measured enrichment value.
measured_enrichment_error	real64	This is the error associated with the measured enrichment value.
wall_thickness	real64	This is the wall thickness value.
thickness_error	real64	This is the error associated with the wall thickness value.
geom_correction	real64	This is the geometric correction factor associated with this source.
wall_material	real64	This is the index of the wall material.

Absorber description keywords

These keywords describe the absorber(s) used in the experiment.

number_of_absorbers	int32	This is the number of absorber layers which are described in the absorber data block.
absorber_material	char* array	This is the description of the absorber material.

absorber_thickness	real64 array	This is the thickness of the absorber material layer.

Collimator description keywords

These keywords describe the collimator(s) used in the experiment

number_of_collimators	int32	This is the number of collimators which are described in the collimator data block.
collimator_type	char * array	Currently acceptable collimator types include *none*, *cylindrical*, and *complex*
collimator_description	char * array	This describes the collimators

Shield Description keywords

These keywords describe the shielding material(s) used in the experiment.

number_of_shields	int32	This is the number of shields that are described in the *shields* data block
shield_type	char * array	This describes the shield type
shield_description	char * array	This describes the shields

Comment keywords

This keyword field allows for user comments.

comments	char *	This is an ASCII string that can include newlines, tabs, etc., in which the user may make additional comments. It is terminated with a null character ('\n').

HDF spectral file rejected keyword data dictionary

There will be occasions that certain keywords and the information they represent will be excluded from the accepted list of keywords. This data dictionary contains a list of the rejected HDF spectral file keywords, their description of the information, and why they were rejected.

count_end_time

It was decided by consensus by the attendees at the LDRD meeting on 12/15 (who also happened to be the initial members of the keyword review committee will be Tom Gosnell (gosnell1@llnl.gov), Zach Koenig (koenig@llnl.gov), Bert Pohl (pohl1@llnl.gov), Dave Knapp (knapp2@llnl.gov), Jim Wolford (wolford@llnl.gov), and Alexis Schach von Wittenau (schach-vonwittenau@llnl.gov) that this field was not necessary since the information that it would have contained can be calculated from other header fields.

RPF Future Directions

Current Status of RPF

As mentioned in the introduction, the RPFTools subroutine library was frozen at version 4.1r3 of HDF. Since that time NCSA has released version 5 of HDF and, if the RPFTools are to be applied to a significant data storage application, it would be wise to upgrade them to be compatible with the latest version of HDF. Because the entire RPFTools library source code comprises only 4500 lines, many of them comment lines, this upgrade would not be arduous.

RPF and Event-Mode Data

It is sometimes desirable to collect nuclear data in event mode. Event mode data are a temporal series of information vectors associated with each detected event such as detector ID, pulse height address, arrival time, GPS coordinates, altitude, temperature, humidity, barometric pressure, etc.

Currently the Radiation Physics Format does not accommodate event-mode data. However, the extensible nature of the RPF means that event mode data can be stored with ease. Since RPF files are self-describing, a variety of event vectors can be accommodated. Should we wish to extend the RPF to include event-mode data, the event vectors will be buffered and then stored to the RPF file as *slabs* in order to increase data collection throughput.

Single Experiment, Multiple Detectors

RPFTools functionality currently supports writing more than one data page per file. The stand-alone RPF application **cdf2rpf** demonstrates this when translating a CDF file which contains a two-part spectrum. The photopeak spectrum and the continuum spectrum are detected by a decrease in the list of bin energies that should be monotonically increasing for a spectrum. It then writes the data into one RPF file as two separate data pages with the same information in the header.

One of the strengths of the RPF file structure lies in its ability to group related data. The challenge lies in determining what is related and what constitutes a group and therefore should be contained in one RPF file. This is dependent upon not only what is being measured but at what level the grouping is to be done.

For example, on a very high level, all the data related to a particular experiment could be considered a group and therefore be written to a single RPF file. But on closer inspection, there could be several different detector types (e.g. NaI, HPGe, neutron, plastic scintillator, temperature, barometric pressure, etc.) and perhaps each detector type should be a group and therefore an RPF file. However, it may make more sense if groups were based upon location and an RPF file would contain data from all co-located detectors.

Determining the optimal grouping of data inside an RPF file must take into consideration not only the type of data that are to be stored but also the eventual use of those data. Ideally, this should be done in a thoughtful way before the data are gathered.

To make the RPF viable for long-term use, a rigid structure must be imposed upon a particular implementation of the RPF data files. Remember that although the RPF files are self describing, the data and tools to manipulate them are very tightly coupled. This means that it must be decided a priori which data will be stored and what keywords will describe it for each detector, experiment, etc., and that any additions or modifications of the RPF structure must be done according to a strict protocol.

This is not as daunting a task as it may seem upon first inspection. It has been successfully done for CDF files. The standalone programs, **cdf2rpf** and **showrpf** are examples of that.

Using RPFTools In a Large-Scale, Multiple-Detector Experiment

The RPF was intended to be adapted to accommodate the copious amounts of data that result from a large-scale, multiple-detector experiment. It just requires a bit of thoughtful planning beforehand to decide which data need to be included and how they should be partitioned. Then the appropriate programs can be written using the RPFTools library to store, retrieve, and manipulate the resulting RPF files.

An RPF data file consists of one or more pages of information. A page consists of a header describing the page plus its associated data. Each page is stored as an HDF *vgroup.*

One scheme establishes the RPF internal *vgroup* hierarchy with the overall experiment at the root of the tree. The next level of interior nodes contains portions of the experiment broken into temporal chunks. This could segregate the members into chunks collected daily, in the morning, or in the afternoon. The members of these temporal *vgroup*s contain portions of the experiment broken into spatial chunks. This would group data from co-located detectors. The leaf nodes of this simple hierarchical tree are data pages describing each detector.

Headers must be established that reflect relevant information at each node level. For example, the header of the root page should contain information describing the overall experiment such as experiment name, date, location, experimenter names, sponsoring organization, etc. The first interior node level should contain information concerning the overall start and stop times of the portion of the experiment described in its members. The second interior node level should contain information concerning the location (e.g. GPS co-ordinates) of the detectors described in its members.

Experimental setup, source location, and other header information may be attached at various levels. It needs to be decided beforehand where it makes most sense.

Currently we have a set of keywords to describe gamma-ray spectra. Sets of keywords to describe neutron, temperature, barometric pressure, and other data need to be developed. It is expected that the neutron keywords will be similar to the gamma keywords. The temperature and barometric keyword sets should be simpler.

Currently we have an established *vdata* to describe gamma-ray spectra. It contains energy bin edges, energy bin centers, and flux values to describe a spectrum. A *vdata* for neutron data should include a temporal index and the count value. The *vdata*s for temperature and barometric pressure would contain a temporal value and the temperature or pressure, respectively.

The data label for the gamma-ray spectra is *gamma_data.* Data labels for the other types of data mentioned would be *neutron_data, temperature,* and *barometric_pressure.*

Appendix A: An Introduction to the Common Data Format (CDF)

Anatomy of a CDF file

The Common Data Format (CDF) was developed in the mid-1980's as a means to store gamma ray spectral data in a manner which would be versatile in both the types and amounts of data to store, easily readable by a human user, and transportable across various computing platforms. The CDF spectral file format is an ASCII format in which can contain more than one page of data. A page of data consists of a header and its associated spectral data.

The header is delimited by *::beginheader::* and *::endheader::* (or *::BEGINHEADER::* and *::ENDHEADER::*, if you really must). Between these delimiting keywords are a user determined number of keyword & value pairs, one pair per line. For example,

FILETYPENO=2

The *FILETYPENO* keyword designates the number of columns in the data section. For backwards/sideways compatibility please be sure to include this in future CDF files. Another keyword to include would be *CHANNELS=*.

The data consist of up to five space-delimited fields of data per line. Type 2 CDF files consist of energy bin edge, flux pairs that appear one pair per line. The first data pair is the leftmost energy bin edge with a dummy flux. The remaining pairs are the right energy bin edge with the flux of that bin. Type 5 CDF files are constructed in the same manner as the Type 2 CDF files with the additional space delimited fields of relative error, absolute error, and original flux on each line.

There can be two-part data: peak data followed by continuum data. This is recognized by a decrease in the normally monotonically increasing energy values.

The CDF file is automatically recognized based upon the presence of the initial keyword *::beginheader::* or *::BEGINHEADER::*.

An example CDF file follows. Not all required keywords are present (e.g. *FILETYPENO*). Also note that Dave Knapp had decided to preface his keywords with an '*'. This is not wrong, just different.

Example CDF file

```
::beginheader::
TITLE=a_file_with_no_name
CHANNELS=        10
TIMECAL= 03-01-96 09:59:46
 slope....: 3.09249E+00
 quad.....: 0.00000E+00
 offset...:-4.24517E+01
```

```
 nchan....:       10
 year.....:     1993
 day......:      173
 time.....:    36047
 livetime.: 2.97000E+02
 realtime.: 3.00000E+02
* TITLE=..\data\U071_414.CNF
* sample_id....: NBS071
* date.........: 14-Mar-1995
* time.........: 14:12:00
* operator.....: John Luke
* location.....: West Dome
* inspector_ID.: Inspectr2120C041
* offset.......: -0.38102504611
* slope........: 0.0769006758928
* livetime.....: 300.000
* realtime.....: 305.410
* deadtime.....: 1.77
* source_matl..: 1
* enrichment...: .711
* enrich_error.: .001
* wall_thick...: 2
* thick_error..: 0.001
* geom_correct.: 1.000
* wall_matl....: 1
* comment_1....: This sample is a 0.711% standard.
* comment_2....:
* comment_3....:
* comment_4....:
 Energy (MeV)   Flux
through the night
15 datapairs in section 1 - 14 bins of data
 7 datapairs in section 2 -  6 bins of data
```

```
::ENDHEADER::
-3.93593E-02   0.00000E+00
1 42
3 44
5 46
7 48
9 50
11 62
13 64
15 66
17 68
```

Sample C code for reading & writing CDF files

The following C code for reading and writing CDF files was stripped out of code written for Palatyi. No attampts were made here to streamline it for simple reading and writing functionality, or to undo the wonderful mung-job that Microsoft Word did on the original formatting. However, it should be fairly straightforward to do so.

read_cdf_file stripped out of getdata.c

```
/*********************************************************************
 * read_cdf_file - read the cdf file
 * 19 aug 1992 clh create
 *  8 sep 1993 clh implement error handling if can't find ::ENDHEADER:: flag.
 * 18 mar 1994 clh incorporate processing for two part cdf file into here.
 *                This has been tested and now works just fine.

*********************************************************************/
void read_cdf_file (d)
char   *d;
{
   viewer_type   *viewer;
   char          r[256], s[256], t[256];   /* strings for filename manipulation. */
   int           i, j;                     /* generic counters. */
   int           havetwo;                  /* if > 0, then have two spectra and the value
                                            is the starting index for the 2nd one. */
   int           tsize;                    /* #edges in the 2nd one. */
```

```
  viewer        = (viewer_type *)d;
  viewer->eflag = 0;
  havetwo       = 0;

  if (viewer->ff_buttons_on == 1)
    free_ff_buttons (viewer);  /* get rid of the button selection display */
  viewer->has_label = 1;
  viewer->wmcolor   = 7;   /* set text color to white */
  viewer->nwmsg     = 1;
  strcpy (r, viewer->gamma_list[viewer->ngammas].fname);   /* hold name root. */
  sprintf (viewer->wmsg[0], "Reading cdf file: %s.", r);
  write_msg (viewer, 57);   /* dark blue gray */
  init_plot_pair (viewer, 0);

  readcdf (viewer->gamma_list[viewer->ngammas].fname,
    &(viewer->gamma_list[viewer->ngammas].odat->nbins),
    &(viewer->gamma_list[viewer->ngammas].odat->e_edge),
    &(viewer->gamma_list[viewer->ngammas].odat->flux),
    &(viewer->gamma_list[viewer->ngammas].odat->d1),
    &(viewer->gamma_list[viewer->ngammas].odat->d2),
    &(viewer->gamma_list[viewer->ngammas].odat->d3), MAX_PEAKS, &viewer-
>eflag);

  if (viewer->eflag == -1)
    {
    viewer->nwmsg = 3;
    sprintf (viewer->wmsg[0], "Error - Error reading cdf file named:");
    sprintf (viewer->wmsg[1], "%s",
      viewer->gamma_list[viewer->ngammas].fname);
    sprintf (viewer->wmsg[0], "::ENDHEADER:: flag was not found.",
      viewer->eflag);
    write_error_msg (viewer, 1);
    return;
```

```
  }

if (viewer->eflag != 0)
  {
  viewer->nwmsg = 3;
  sprintf (viewer->wmsg[0], "Error - Error reading cdf file named:");
  sprintf (viewer->wmsg[1], "%s",
    viewer->gamma_list[viewer->ngammas].fname);
  sprintf (viewer->wmsg[0], "Found %d lines of invalid data.",
    viewer->eflag);
  write_error_msg (viewer, 1);
  }

viewer->gamma_list[viewer->ngammas].odat->asize =
  viewer->gamma_list[viewer->ngammas].odat->nbins;

/* to check for two spectra in one file, see if we can spot a decrease in the
  supposedly monotonically increasing energy. */
for (i = 0; i < (viewer->gamma_list[viewer->ngammas].odat->nbins - 1); i++)
  {
  if (viewer->gamma_list[viewer->ngammas].odat->e_edge[i+1] <
    viewer->gamma_list[viewer->ngammas].odat->e_edge[i])
    {
   /* we found another one so fix the #things in the first one and the filename. */
    havetwo = i + 1;
    viewer->gamma_list[viewer->ngammas].odat->nbins = havetwo - 1;
    strcpy (s, viewer->gamma_list[viewer->ngammas].fname);   /* hold name root. */
    strcpy (t, viewer->gamma_list[viewer->ngammas].fname);   /* hold name root. */
    strcat (s, ".peak");
    strcat (t, ".cntuum");
    free (viewer->gamma_list[viewer->ngammas].fname);
    viewer->gamma_list[viewer->ngammas].fname =
          (char *) malloc (sizeof(char) * (strlen(s)+1));
```

```
        strcpy (viewer->gamma_list[viewer->ngammas].fname, s);
        break;
        }
      }

  /* calculate the bin centers and the energy & flux bounds for the first one. */
  convert_to_center (viewer->gamma_list[viewer->ngammas].odat->nbins+1,
    viewer->gamma_list[viewer->ngammas].odat->e_edge,
    &(viewer->gamma_list[viewer->ngammas].odat->e_center));

  efbounds (viewer->gamma_list[viewer->ngammas].odat->e_edge,
    viewer->gamma_list[viewer->ngammas].odat->nbins + 1,
    &viewer->gamma_list[viewer->ngammas].odat->enmin,
    &viewer->gamma_list[viewer->ngammas].odat->enmax);
  efbounds (viewer->gamma_list[viewer->ngammas].odat->flux,
    viewer->gamma_list[viewer->ngammas].odat->nbins,
    &viewer->gamma_list[viewer->ngammas].odat->flmin,
    &viewer->gamma_list[viewer->ngammas].odat->flmax);

#undef DEBUG
#ifdef DEBUG
  print_struct_specdat_type (viewer->gamma_list[viewer->ngammas].odat, 1);
#endif

  /* set the other parameters for the first one. */
  viewer->gamma_list[viewer->ngammas].fmt_id  = gdf_cdf;
  viewer->gamma_list[viewer->ngammas].visible = 1;
  viewer->gamma_list[viewer->ngammas].color   = return_next(MAX_WAVE_COLORS,
    viewer->used_colors);
  viewer->used_colors[viewer->gamma_list[viewer->ngammas].color]++;
  viewer->nwmsg = 1;
  sprintf (viewer->wmsg[0], "Finished reading CDF file: %s.", r);
  write_msg (viewer, 57);   /* dark blue gray */
```

```
viewer->ngammas++;   /* so we don't step on what we just made. */

/* if we have a second one make space for the 2nd spectrum and fill it. */
if (havetwo > 0)
  {
  viewer->gamma_list[viewer->ngammas].fname =
        (char *) malloc (sizeof(char) * (strlen(t)+1));
  strcpy (viewer->gamma_list[viewer->ngammas].fname, t);

  tsize = viewer->gamma_list[viewer->ngammas-1].odat->asize
        - viewer->gamma_list[viewer->ngammas-1].odat->nbins;
  init_plot_pair (viewer, tsize);

  for (j = 0; j < tsize; j++)
    {
    viewer->gamma_list[viewer->ngammas].odat->e_edge[j] =
      viewer->gamma_list[viewer->ngammas-1].odat->e_edge[j+havetwo];
    viewer->gamma_list[viewer->ngammas].odat->flux[j] =
      viewer->gamma_list[viewer->ngammas-1].odat->flux[j+havetwo];
    }

  viewer->gamma_list[viewer->ngammas].odat->asize = tsize - 1;
  viewer->gamma_list[viewer->ngammas].odat->nbins = tsize - 1;

  convert_to_center (viewer->gamma_list[viewer->ngammas].odat->nbins+1,
    viewer->gamma_list[viewer->ngammas].odat->e_edge,
    &(viewer->gamma_list[viewer->ngammas].odat->e_center));

  efbounds (viewer->gamma_list[viewer->ngammas].odat->e_edge,
    viewer->gamma_list[viewer->ngammas].odat->nbins + 1,
    &viewer->gamma_list[viewer->ngammas].odat->enmin,
    &viewer->gamma_list[viewer->ngammas].odat->enmax);
  efbounds (viewer->gamma_list[viewer->ngammas].odat->flux,
```

```
        viewer->gamma_list[viewer->ngammas].odat->nbins,
        &viewer->gamma_list[viewer->ngammas].odat->flmin,
        &viewer->gamma_list[viewer->ngammas].odat->flmax);

#undef DEBUG
#ifdef DEBUG
  print_struct_specdat_type (viewer->gamma_list[viewer->ngammas].odat, 1);
#endif

      viewer->gamma_list[viewer->ngammas].fmt_id  = gdf_cdf;
      viewer->gamma_list[viewer->ngammas].visible = 1;
      viewer->gamma_list[viewer->ngammas].color   =
return_next(MAX_WAVE_COLORS,
        viewer->used_colors);
      viewer->used_colors[viewer->gamma_list[viewer->ngammas].color]++;
      viewer->nwmsg = 2;
      sprintf (viewer->wmsg[0], "Finished splitting CDF file: %s.", r);
      sprintf (viewer->wmsg[1], "   into two spectra.",
        viewer->gamma_list[viewer->ngammas].fname);
      write_msg (viewer, 57);   /* dark blue gray */
      viewer->ngammas++;   /* so we don't step on what we just made. */
      }

  if (viewer->after_uget == 1)
      {
      add_curve ((char *)viewer);
      }

#undef DEBUG
} /* read_cdf_file */
```

write_cdf_waveform stripped out of viewer.c

```
/
***************************************************************************
*******
 * write_cdf_waveform() writes wave data to an ascii CDF file.
 * A CDF header is followed by bin edges & flux pairs.
 * 15 jun 1993 clh create from write_waveform
 * 24 aug 1993 clh change CDF output file data format; add FILETYPENO & TIMECAL

***************************************************************************
********/
void
write_cdf_waveform (w_id, d, e_id)
window_type     *w_id;
char                  *d;
event_data_type          *e_id;
{
   int                  i, iwhich;
   viewer_param_type   *par;
   viewer_type        *viewer;
   char                oname[256];
   char                s[256];
   FILE                *fp;
   char                dstring[20];
   struct tm           *tp;
   time_t              clock;

   par    = (viewer_param_type *)d;
   viewer = par->viewer;
   iwhich = par->number;

   i = 0;
   sprintf (oname, "%s.cdf_%d", viewer->gamma_list[iwhich].fname, i);

   /* let's make sure we don't nuke anything important! */
   while (!access(oname, F_OK))
```

```
  {
  sprintf (oname, "%s.cdf_%d", viewer->gamma_list[iwhich].fname, i++);
  }
if ((fp = fopen (oname, "w")) == NULL)
  {
  viewer->nwmsg = 2;
  sprintf (viewer->wmsg[0], "Error - Cannot open the file named:");
  sprintf (viewer->wmsg[1], " %s", oname);
  write_error_msg (viewer, 0);
  }
 else
  {
  viewer->ncmsg = 3;
  sprintf (viewer->cmsg[0], "Writing CDF file:");
  sprintf (viewer->cmsg[1], " %s.", oname);
  sprintf (viewer->cmsg[0], "bin edge & flux pairs.");
  coef_msg (viewer);
  fprintf (fp, "::BEGINHEADER::\n");
  /* the filename */
  fprintf (fp, "TITLE= %s\n", viewer->gamma_list[iwhich].fname);
  /* when you wrote this CDF file */
  clock = time(0);
  tp    = localtime (&clock);
  sprintf (dstring, "%02.2d-%02.2d-%02.2d %02.2d:%02.2d:%02.2d",
    tp->tm_mon+1, tp->tm_mday, tp->tm_year, tp->tm_hour, tp->tm_min, tp->tm_sec);
  fprintf (fp, "TIMECAL= %s\n", dstring);
  /* how many columns of data the CDF file has (2 = energy & flux) */
  fprintf (fp, "FILETYPENO=      %d\n", 2);
  /* how many channels of data the CDF file has */
  fprintf (fp, "CHANNELS=       %d\n", viewer->gamma_list[iwhich].odat->nbins);
  fprintf (fp, "::ENDHEADER::\n");
  fprintf (fp, "%13.6e %13.6e\n", viewer->gamma_list[iwhich].odat->e_edge[0], 0.0);
  for (i = 0; i < viewer->gamma_list[iwhich].odat->nbins; i++)
```

```
        {
        fprintf (fp, "%13.6e %13.6e\n",
          viewer->gamma_list[iwhich].odat->e_edge[i+1],
          viewer->gamma_list[iwhich].odat->flux[i]);
          }
    fclose (fp);
    viewer->ncmsg = 5;
    sprintf (viewer->cmsg[0], "Click on the output type to");
    sprintf (viewer->cmsg[1], "write the waveform to a file.");
    sprintf (viewer->cmsg[2], " ");
    sprintf (viewer->cmsg[3], "Wrote CDF file:");
    sprintf (viewer->cmsg[4], " %s.", oname);
    coef_msg (viewer);
    }
} /*
```

Appendix B: About HDF and supported platforms

In this appendix are a few web pages downloaded from the National Center for Supercomputing Applications (NCSA) web site, *http://hdf.ncsa.uiuc.edu/*, that define the Hierarchical Data Format (HDF) and list the platforms (with compiler information) on which the DCSA HDF group tested HDF and for which pre-compiled binaries are provided on the NCSA ftp server.